

# Интерфейсы

Лекция №8

## *Синтаксис интерфейса*

Интерфейс предназначен для определения характеристик и поведения, присущих классам, реализующим этот интерфейс.

В интерфейсе задается набор методов, свойств и индексаторов, которые должны быть реализованы в производных классах.

```
[<спецификаторы>] interface <имя > [: предки]  
    {<тело интерфейса>}
```

Спецификаторы – это **public** или **internal** (по умолчанию).

Для вложенных в класс интерфейсов можно использовать спецификаторы **new**, **protected**, **private**.

Существует традиция имя интерфейса начинать с префикса **I** (но это не является обязательным).

Предки – это интерфейсы, элементы которых наследует данный интерфейс. Имена предков перечисляются через запятую.

Тело интерфейса составляют заголовки методов, шаблоны свойств и индексаторов, события.

Заголовки методов объявляются следующим образом:

**<тип\_результата>   <имя\_метода> (<список\_параметров>) ;**

В интерфейсе методы неявно являются открытыми (**public**), при этом не разрешается **явным образом** указывать спецификатор доступа.

Шаблон свойства представляется следующим образом:

```
< тип свойства>   <имя свойства>  
{ get ; set ;}
```

Свойство может быть только для чтения **{ get ;}** или только для записи **{ set ;}**

Шаблон индексатора (одномерного) имеет вид:

```
< тип результата>   this[<тип индекса> <имя индекса>]  
{ get ; set ;}
```

Интерфейсы не могут иметь константы и поля. Они не могут определять конструкторы, деструкторы, операции.

Ни один член интерфейса не может быть объявлен статическим.

Например, рассмотрим интерфейс, определяющий некоторые характеристики объектов живой природы.

```
interface IOpisanie
{
    double Ves { get;}    //шаблоны свойств
    double Rost{get;}    // только для чтения
    int Vozrast( );      // метод
    void Vyvod(string name); // метод
}
```

## Реализация интерфейсов

Интерфейс может быть реализован любым количеством классов.

При этом один класс может реализовать любое число интерфейсов.

Чтобы реализовать интерфейс, нужно указать его имя после имени класса:

```
class <имя_класса> : <имя__интерфейса>  
  
{<тело класса>}
```

Класс, реализующий интерфейс, должен определять все элементы этого интерфейса.

В классах, реализующих интерфейс, можно также определять дополнительные элементы, не входящие в интерфейс.

Методы, которые реализуют интерфейс, должны быть объявлены **открытыми**.

Сигнатура в реализации метода должна в точности совпадать с сигнатурой метода, заданной в определении интерфейса.

Например, рассмотрим класс Chelovek, реализующий приведенный ранее интерфейс IOpisanie.

```
class Chelovek : IOpisanie
{
    protected string fam;
    double ves, rost;
    public int god_rogd;

    public Chelovek(string f, double v, double r, int g_r)
        { fam = f; ves = v; rost = r; god_rogd = g_r; }
```

```
public double Ves {get {return ves;}}  
public double Rost { get { return rost; } }  
  
public int Vozrast( )  
    { return DateTime.Now.Year - god_rogd; }  
  
public void Vyvod(string name)  
    { Console.WriteLine(name + " " + fam + " "+Vozrast( )); }  
  
public int Vozrast(int g) { return g - god_rogd; }  
}
```

Тогда, например, в методе Main, можно следующим образом работать с объектом класса:



```
Chelovek  men = new  Chelovek("Васечкин", 89, 180, 1975);  
men.Vyvod("бизнесмен");
```

```
Console.WriteLine("его год рождения:"+men.god_rogd+  
" в 2000 г. ему было "+men.Vozrast(2000));
```

Можно создавать объекты типа интерфейс следующим образом:

```
<имя интерфейса>  <имя объекта> =  
new  <конструктор класса, реализующего интерфейс>;
```

Например,

```
IOpisanie men1 = new Chelovek("Петров",65,165,1995);
```

В этом случае созданный объект имеет доступ только к элементам интерфейса.

Например, можно

```
men1.Vyvod("школьник");
```

Но нельзя использовать обращение

```
men1.god_rogd или men1.Vozrast(2000)
```

Этот же интерфейс **IOpisanie** может быть реализован другим классом, например, классом **Sobaka**.

```
class Sobaka : IOpisanie
```

```
{ string klichka;
```

```
double ves, holka;
```

```
string dat_rogd; // дата в формате ДД.ММ.ГГГГ
```

```
int[ ] schenki;
```

```
public Sobaka(string imja, double v,double h,
```

```
int n,int[ ] k_sch,string d_r)
```

```
{
```

```
klichka = imja; ves = v; holka = h; dat_rogd = d_r;
```

```
schenki = new int[n]; schenki = k_sch; }
```

```
public double Ves { get { return ves; } }
```

```
public double Rost { get { return holka; } }
```

```
public int Vozrast( )
```

```
{ return
```

```
    DateTime.Now.Year -
```

```
    Convert.ToInt32(dat_rogd.Substring(dat_rogd.Length-4, 4));
```

```
}
```

```
public void Vyvod(string poroda)
```

```
{ Console.WriteLine(poroda + " " + klichka + " " + Vozrast( ));
```

```
}
```

```
public void Vyvod( )  
    { Console.WriteLine("Количество щенков:");  
    for(int i=0; i<schenki.Length; i++)  
        Console.WriteLine((i+1) + "-й раз:" + schenki[i]);  
    }  
}
```

Тогда с объектом класса `Sobaka` можно работать так:

```
int[ ] sch={3,2};  
Sobaka Taksa =  
    new Sobaka("Жучка", 9, 20, 2, sch, "12.11.2005");  
Taksa.Vyvod("Такса"); Taksa.Vyvod();
```

Классы могут реализовать несколько интерфейсов. В этом случае имена интерфейсов отделяются запятыми.

Например, создадим еще один интерфейс:

```
interface IObrabotka
{
    double Rezult( );
    bool Bolshe (IObrabotka S),
    int this[int i] { get;}
}
```

В данном случае в качестве аргумента можно использовать объект любого класса, реализующего данный интерфейс

И пусть класс Sobaka реализует его наряду с интерфейсом IOpisanie.

Тогда заголовок класса должен выглядеть следующим образом:

```
class Sobaka : IOpisanie, IObrabotka
```

А класс нужно дополнить реализацией элементов интерфейса IObrabotka.

```
public double Rezult( ) // общее количество щенков
```

```
{ int s = 0; foreach (int sch in schenki) s = s + sch;  
  return s; }
```

```
public int this[int i] { get { return schenki[i]; } }
```

```
public bool Bolshe(IObrabotka S)
{
    if (Rezult( ) > S.Rezult( ))    return true;
    else    return    false;
}
```

Тогда, например, если **Mops** — объект класса **Sobaka**, допустим такой оператор:

```
if (Taksa.Bolshe(Mops))
    Console.WriteLine("У таксы больше щенков чем у мопса");
else
    Console.WriteLine("У таксы не больше щенков чем у мопса");
```



Класс может наследовать базовый класс и реализовать один или несколько интерфейсов. В этом случае список интерфейсов должно возглавлять имя базового класса.

**class <имя\_класса> :**

**<имя базового класса>, <имя\_\_интерфейса\_1>,**

**<имя\_\_интерфейса\_2>, и т.д.**

Если базовый класс реализовывал какой-либо интерфейс, то производный класс также является реализующим интерфейс.

Например, пусть класс Student Является производным от Chelovek.

```
class Student : Chelovek
```

```
{ int[ ] ochenki = new int[3];
```

```
public Student(string f, double v, double r, int g_r, int[ ] oc)
```

```
:base(f,v,r,g_r)
```

```
{ ochenki = oc; }
```

```
new public void Vyvod(string name)
```

```
{ base.Vyvod(name);
```

```
Console.WriteLine("Оценки за сессию:");
```

```
foreach (int o in ochenki) Console.WriteLine(o);
```

```
}
```

```
}
```

Тогда, чтобы убедиться, что объекты этого класса поддерживают данный интерфейс, можно применить следующий код:

```
int[ ] o1 = { 2,2,2};
```

```
IOpisanie TheBad = new Student("Тимоти",90,190,1980,o1);
```

```
TheBad.Vyvod("студент");
```

Если понадобится, чтобы студенты поддерживали интерфейс IObrabotka, нужно добавить этот интерфейс в заголовок класса:

```
class Student:Chelovek, IObrabotka
```

и дополнить класс Student реализацией элементов этого интерфейса:

```
public double Rezult( ) //средний балл
```

```
{ int s = 0; foreach (int oc in ocenki) s = s + oc; return s/3; }
```

```
public int this[int i] { get { return ocenki[i]; } }
```

```
public bool Bolshe(IObrabotka S)
```

```
{
```

```
    if (Rezult( ) > S.Rezult( )) return true;
```

```
    else return false;
```

```
}
```

Теперь класс Student поддерживает два интерфейса.

При этом появилась возможность сравнить объект класса Student с объектом класса Sobaka.

например, пусть

```
Student TheBad = new Student("Тимоти",90,170,1980,01);
```

Тогда допустим такой оператор:

```
if (TheBad.Bolshe(Taksa))
```

```
    Console.WriteLine(" Тимоти лучше таксы");
```

```
else
```

```
    Console.WriteLine("Такса не хуже Тимоти");
```

При реализации в классе элемента интерфейса можно в заголовке явно указывать имя интерфейса перед именем метода, свойства или перед словом **this** индексатора без указания спецификатора доступа :

**<тип результата> <имя интерфейса >.<имя метода>  
(<список параметров>)**

**<тип результата> <имя интерфейса >.this[<тип> <индекс>]**

К этим элементам можно обращаться только от имени объекта типа этого интерфейса.

Например, пусть в классе `Sobaka` индексатор описан следующим образом:

```
int  IObrabotka.this[int i] { get { return schenki[i]; } }
```

Тогда следующий фрагмент приведет к ошибке:

```
int[ ] sch1 = {3,2,5};
```

```
Sobaka Mops = new Sobaka("Тузик", 20, 32, 3, sch1, "13.12.2000");
```

```
Console.WriteLine("У Тузика 1-й раз было щенков "+Mops[0]);
```

Правильно будет:

```
Console.WriteLine("У Тузика 1-й раз было щенков "+  
((IObrabotka)Mops)[0]);
```

ИЛИ

```
IObработка Mops = new Собака("Тузик", 20, 32, 3, sch1, "13.12.2000");  
Console.WriteLine("У Тузика 1-й раз было щенков "+Mops[0]);
```

Явное указание имени интерфейса может быть полезным при множественном наследовании интерфейсов, когда в разных интерфейсах встречаются методы с одинаковыми именами.

### *Операции is и as.*

Бинарная операция **is** определяет, совместим ли текущий тип объекта, расположенного слева от **is**, с типом, указанным справа.



**<объект> is <тип>**

Например: **x is Array**

Результат операции равен **true**, если объект можно преобразовать к заданному типу, и **false** в противном случае.

Результат операции **Taksa is Student - false.**

Результат операции **Taksa is IOpisanie - true.**

Эту операцию можно использовать, чтобы проверить, поддерживает ли аргумент, используемый вместо параметра типа **object**, нужный интерфейс.

Например, метод **Bolshe** может быть следующим :

```
public bool Bolshe(Object S)
{
    if (S is IObrabotka)
    {
        if (Rezult( ) > ( (IObrabotka) S).Rezult( )) return true;
        else return false;
    }
    else
    {
        if (S is IOpisanie)
        {
            if (Rost > ((IOpisanie)S).Rost) return true; else return false; }

        else throw new Exception("Несравнимые величины");
    }
}
```

Тогда будем иметь право на использование такого оператора:

```
if (TheBad.Bolshe(men))
```

```
    Console.WriteLine(" Тимоти круче Васечкина");
```

```
else
```

```
    Console.WriteLine(" Тимоти не круче Васечкина");
```

Операция **as** выполняет преобразование к указанному типу, если это невозможно формирует результат **null**.

**<объект> as <тип>**

Например, в метод **Bolshe** можно внести следующие изменения:

```
IOpisanie SS=S as IOpisanie;
```

```
    if (SS!=null)
```

```
    { if (Rost > SS.Rost) return true;
```

```
                                             else return false;
```

```
    }
```

```
    else throw new Exception("Несравнимые величины");
```

## **Стандартные интерфейсы .NET**

Интерфейс **Comparable** определен в пространстве имен System.

Содержит один метод, предназначенный для сравнения текущего объекта и объекта, переданного в качестве параметра:

**int CompareTo( object obj)**

Результат работы этого метода должен быть таким:

0 – если вызывающий объект и параметр равны,

*положительное число*, если вызывающий объект больше параметра obj,

*отрицательное число*, если вызывающий объект меньше параметра obj.

Если тип объекта *obj* несовместим с вызывающим объектом, метод генерирует исключение типа **ArgumentException**.

Реализовывать этот интерфейс в классе необходимо, чтобы использовать стандартные средства языка для работы с объектами, для которых определена процедура сравнения.

Например, пусть St – массив объектов класса Student:

```
Student[ ] St = new Student[25];
```

Попытка выполнить сортировку с помощью метода класса Array -

```
Array.Sort(St);
```

приведет к ошибке.

Даже определив в классе метод с такой же сигнатурой, воспользоваться методом `Sort` будет нельзя.

Если же в заголовке класса `Student` в список интерфейсов добавить интерфейс **`Comparable`** и реализовать его метод в классе, можно без проблем использовать метод **`Array.Sort`**

Заголовок примет вид:

```
class Student: Chelovek, IObrabotka, Comparable
```

Пример реализации метода:

```
public int CompareTo(object obj)
{ Student Stud = obj as Student;
  if (Stud != null)
  {
    if (Ves == Stud.Ves) return 0;
    else if (Ves > Stud.Ves) return 1;
    else return -1;
  }
  else throw new ArgumentException();
}
```



Интерфейс **IComparer** определен в пространстве имен **System.Collections**.

Основное достоинство использования интерфейса **IComparer** состоит в том, что он позволяет сортировать объекты классов по различным критериям, кроме того позволяет сортировать объекты классов которые не реализуют интерфейс **IComparer**.

Содержит один метод, предназначенный для сравнения двух объектов, переданных в качестве параметров:

```
int Compare ( object obj1, object obj2)
```

Результат работы этого метода должен быть таким:

0 – если параметры равны,

*положительное число*, если первый объект больше второго,

*отрицательное число*, если первый объект меньше второго.

Для каждого критерия сортировки необходимо создать класс, реализующий интерфейс **IComparer**.

При сортировке объект такого класса можно использовать в качестве второго параметра перегруженного метода **Sort**

Например, если нужно отсортировать массив студентов по фамилии, можно создать такой класс:

```
class SortFam : IComparer
{
    public int Compare(object ob1, object ob2)
    {
        Student sob1 = (Student)ob1;
        Student sob2 = (Student)ob2;
        return String.Compare(sob1.fam, sob2.fam);
    }
}
```

Тогда после выполнения оператора

**Array.Sort(St, new SortFam( ));**

массив студентов будет отсортирован по фамилии.

При этом класс Student не поддерживает интерфейс **IComparer**.

Интерфейс **IEnumerable** определен в пространстве имен System.Collections.

Содержит один метод:

**IEnumerator GetEnumerator( );**

Этот метод возвращает **нумератор (перечислитель)** – объект интерфейсного типа **IEnumerator**, который можно использовать для просмотра элементов объекта реализующего класса.

Интерфейс **IEnumerator** определяет три элемента:

- свойство **object Current {get;}**, которое возвращает текущий элемент объекта;
- метод **bool MoveNext( )**, продвигающий перечислитель на следующий элемент объекта;

Метод возвращает значение **true**, если к следующему элементу можно получить доступ, или значение **false** в противном случае.

До выполнения первого обращения к методу **MoveNext ( )** значение свойства **Current** не определено.

- метод **void Reset( )**, устанавливающий перечислитель в начало просматриваемого объекта.

После вызова метода **Reset( )** для доступа к первому элементу необходимо вызвать метод **MoveNext ( )**.

Например, для одномерного массива (т.к. в классе `Array` реализованы интерфейсы **`IEnumerable`** и **`IEnumerator`**) можно так использовать нумератор:

```
int[ ] x = { 1,3,5,7};
```

```
IEnumerator xx = x.GetEnumerator( );
```

```
xx.Reset( );
```

```
while (xx.MoveNext( ))
```

```
    Console.WriteLine(xx.Current);
```

Цикл **`foreach`** использует методы интерфейса **`IEnumerator`** для перебора элементов. Поэтому описанный фрагмент можно записать так:

```
foreach (int xt in x) Console.WriteLine(xt);
```

Таким образом, для того, чтобы иметь возможность применять оператор `foreach` для перебора элементов объекта, нужно в классе реализовать три метода (**GetEnumerator**, **MoveNext**, **Reset**) и одно свойство **Current**.

Опишем для примера класс студентов, реализующий эти элементы.

```
class Student: IEnumerable, IEnumerator
```

```
{ string fam;
```

```
    double sb;
```

```
    int gr;
```

```
        int nom; \\ вспомогательная переменная для  
        \\определения номера поля, к которому перечислитель  
        \\получает доступ
```

```
public Student(string f, double ss, int god)
```

```
{ fam = f; sb = ss; gr = god;}
```

```
public bool MoveNext( )
```

```
{nom = nom + 1;
```

```
    if (nom < 3) return  true;
```

```
    else return  false;
```

```
}
```

```
public void Reset( ) { nom = -1; }
```



public object Current

{ get

{ switch (nom)

{ case 0: return fam;

case 1: return sb;

case 2: return gr;

default: return -1;

}

}

}

```
public IEnumerator GetEnumerator()  
    { this.Reset( );  
      return this; }  
}
```

Тогда в программе допустимы следующие операторы:

```
Student S = new Student("Иванов",7.8,1989);  
  
foreach (object z in S) Console.WriteLine(z);
```

Результат: **Иванов**

**7,8**

**1989**

Можно так:

```
S.Reset();  
while (S.MoveNext( ))  
    { Console.WriteLine(S.Current); }
```

Можно не реализовывать интерфейс **IEnumerator**, а использовать **итератор**.

**Итератор** — это фрагмент кода, задающий последовательность перебора элементов объекта.

Порядок получения доступа к элементам определяется конструкцией

**yield return**

которая формирует значение, выдаваемое на очередной итерации.

Итератор может использоваться в теле метода, операции или в get-аксессоре свойства, если эти элементы класса возвращают значение типа **IEnumerator** или **IEnumerable**.

Например, пусть класс студентов реализует только интерфейс **IEnumerable**.

```
class Student: IEnumerable
```

Метод `GetEnumerator( )` с помощью итератора можно реализовать следующим образом:

```
public IEnumerator GetEnumerator( )  
    { yield return fam;  
      yield return sb;  
      yield return gr;}
```

Тогда допустимы следующие операторы:

```
Student S = new Student("Иванов",7.8,1989);  
foreach (object z in S) Console.WriteLine(z);
```

```
IEnumerator Z = S.GetEnumerator( );  
Z.MoveNext();      Console.WriteLine(Z.Current);
```

Итератор выполняется следующим образом:

Компилятор создает служебный объект-перечислитель.

При вызове для него метода `MoveNext( )` выполняются операторы, расположенные до первой конструкции **yield** **return** и выдается соответствующее значение, а выполнение итератора прерывается.

Следующий вызов метода `MoveNext( )` объекта-перечислителя возобновляет выполнение итератора с того момента, на котором оно было остановлено в прошлый раз, снова выполняются операторы до **yield** **return** и выдается очередное значение.

Таким образом, итератор выполняется не весь последовательно, а разбит на отдельные итерации, между выполнением которых состояние итератора сохраняется.

Итератор может выглядеть так:

```
for (int i = 0; i < n; i++) yield return m[i];
```

С помощью итераторов можно организовать различный порядок перебора элементов для одного и того же класса.

Например, можно организовать перебор элементов в объекте класса студент в обратном порядке:

```
public IEnumerable Back( )  
    { yield return gr;  
      yield return sb;  
      yield return fam;  
    }
```

Тогда выполнение следующих операторов:

```
Student S = new Student("Иванов",7.8,1989);
```

```
foreach (object z in S.Back( )) Console.WriteLine(z);
```

приведет к результату:

1989

7,8

Иванов



## Использование операции *typeof*

Формат операции:

**typeof(<тип>)**

Результатом этой операции является объект класса **System.Type**, содержащий информацию, связанную с заданным типом.

Для объекта типа **Type** можно использовать различные свойства, поля и методы, определенные в классе **Type**.

Например, `Type T = typeof(double);`

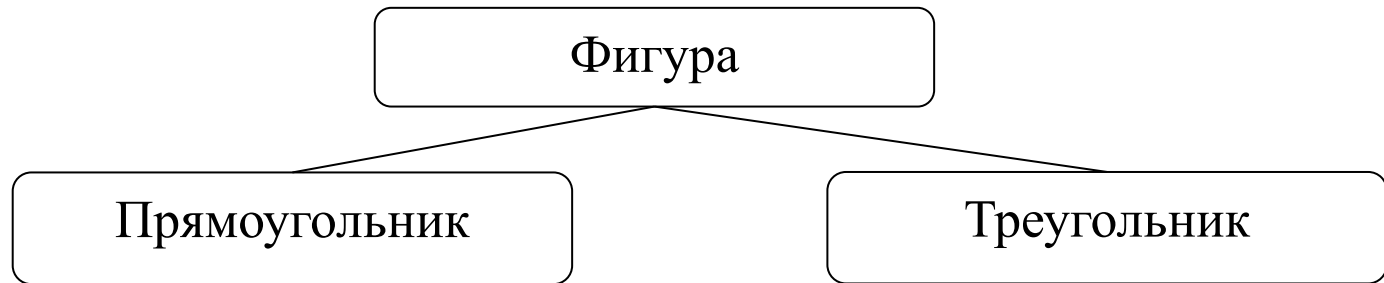
```
Console.WriteLine(T.FullName+" "+T.IsInterface+" "+T.IsClass+  
                  " "+T.IsPrimitive);
```

**Результат:**

<b>System.Double</b>	<b>False</b>	<b>False</b>	<b>True</b>
----------------------	--------------	--------------	-------------

## Пример.

Создать иерархию классов:



Класс «Фигура» должен быть *абстрактным*, содержать следующие элементы: поля (массив, содержащий длины сторон; цвет фигуры); абстрактный метод вычисления периметра; метод вывода информации об объекте. Этот класс должен реализовывать интерфейс **Comparable**.

Классы прямоугольников и треугольников должны содержать переопределенные методы для вычисления периметра.

Разработать программу, которая выполняет следующие действия:

- считывает информацию из текстового файла, каждая строка которого содержит длины сторон прямоугольника или треугольника и цвет фигуры, например:

2 4 Red

1 3 3 White

3 6 4 Red

3 5 Yellow

- формирует на основании этой информации массив объектов базового класса иерархии;
- выводит на экран всю информацию в виде:

Номер	Вид фигуры	Периметр	Цвет
1	треугольник	15,23	Red

При этом каждая строка выводится тем цветом, который указан в графе цвет.

- сортирует массив в порядке возрастания площадей многоугольников и выводит отсортированный массив.

```
abstract class Gfig : IComparable
```

```
{    protected double[ ] storona;
```

```
    protected string color;
```

```
    abstract protected double perimetr( );
```

```
public void vyvod(int i)
```

```
{    string s;
```

```
    if (storona.Length == 2)    s="прямоугольник";
```

```
    else
```

```
        {if(storona.Length == 3) s="треугольник";
```

```
            else                s="неизвестно";}
```

```
Console.ForegroundColor =
```

```
(ConsoleColor) Enum.Parse(typeof(ConsoleColor), color);
```

```
Console.WriteLine("{0,2} {1,15} {2,7:f2} {3}",i,s,perimetr(),color); }
```

```
public int CompareTo(Object obj)
```

```
{    Gfig ff = obj as Gfig;
```

```
    if (perimetr( ) > ff.perimetr( ))    return 1;
```

```
    else
```

```
        { if (perimetr( ) < ff.perimetr( ))    return -1;
```

```
        else    return 0; }
```

```
    } }
```

```
class P : Gfig
```

```
{
```

```
    public P(double[] x, string cl)
```

```
    {
```

```
        storona = new double[2];
```

```
        storona = x; color = cl; }
```

```
protected override double perimetr()
```

```
    { return 2*(storona[0] + storona[1]); }
```

```
}
```

```
class T : Gfig
```

```
{
```

```
    public T(double[ ] x, string cl)
```

```
    {
```

```
        storona = new double[3];
```

```
        storona = x; color = cl;
```

```
    }
```

```
protected override double perimetr()
```

```
    { return (storona[0] + storona[1] + storona[2]);
```

```
    }
```

```
}
```



В методе Main:

```
StreamReader f = new StreamReader("Figury.txt");
```

```
string s = f.ReadLine( );    int j = 0;
```

```
while (s != null)
```

```
{
```

```
    s = f.ReadLine();
```

```
    j++;
```

```
}
```

```
string[ ] dano = new string[j];
```

```
f.Close( );
```

```
f = new StreamReader("Figury.txt");
```

```
s = f.ReadLine( ); j = 0;
```

```
while (s != null)
```

```
{
```

```
    dano[j] = s;
```

```
    s = f.ReadLine( );
```

```
    j++;
```

```
}
```

```
f.Close( );
```

```
Gfig[ ] fig = new Gfig[dano.Length];

for (int i=0; i<dano.Length; i++)
{
    string[ ] ss = dano[i].Split( );
    if (ss.Length == 4)
    {
        double[ ] x = new double[3];

        for (int jj = 0; jj < 3; jj++)    x[jj] = Convert.ToDouble(ss[jj]);

        fig[i] = new T(x, ss[3]);
    }
    else
```

```
{  
    double[ ] x = new double[2];  
    for (int jj = 0; jj < 2; jj++)  
        x[jj] = Convert.ToDouble(ss[jj]);  
    fig[i] = new P(x, ss[2]);  
}  
}
```

```
for (int i = 0; i < fig.Length; i++)  
    { fig[i].vyvod(i); }
```

```
Array.Sort(fig);
```

```
Console.WriteLine("Отсортированный массив:");
```

```
for (int i = 0; i < fig.Length; i++)
```

```
    { fig[i].vyvod(i); }
```

```
Console.ReadKey( );
```

```
    }
```

```
}
```